Hedgehog: A Performance-Oriented General-Purpose Library for Multi-GPU Systems

Alexandre Bardakoff – Timothy Blattner
Bruno Bachelet – Walid Keyrouz – Loic Yon

National Institute of Standards and Technology
U.S. Department of Commerce

Université Clermont Auvergne

LIMOS

Information Technology Laboratory

Disclaimer

No approval or endorsement of any commercial product by NIST is intended or implied. Certain commercial software, products, and systems are identified in this report to facilitate better understanding. Such identification does not imply recommendations or endorsement by NIST, nor does it imply that the software and products identified are necessarily the best available for the purpose.

Acknowledgment

- NIST
  - Mary Brady
  - Walid Keyrouz
- LIMOS
  - Bruno Bachelet
  - Loïc Yon

Motivation – Hardware

- Servers
  - AMD EPYC 7702P w/**64 cores,** Intel Xeon Platinum 8253 Processor w/**16 cores**
- Desktops
  - AMD Ryzen Threadripper 3990X w/**64 cores,** AMD Ryzen 9 PRO 3900 w/**12 cores**
  - Intel Core i9-10980XE Extreme Edition w/**18 cores (3x hyperthreading)**
- Laptops
  - AMD Ryzen 7 4800H w/**8 cores,** Intel Core i9-9980HK w/**8 cores**
- Mobile CPU: Kryo 585 w/**8 cores**
- GPUs:
  - GeForce RTX 2080: **9362** (SP), **292.6** (DP), **18720** (HP) **GFLOPS**
  - Tesla T4 GPU accelerator: **8100** (single precision) **GFLOPS**

Motivation – Understandable Scalable Programs

- Abstract model of execution
- Explicit representation of an algorithm
  - Exists during execution
  - Used to instrument and reason about performance
- Experimentation for performance using high-level abstractions
  - Without loss of potential performance

Requirements

- Manage a node with many cores and one or multiple GPUs
- Explicit representation of an algorithm (that exists during execution)
- High-level abstractions (without loss of potential performance)

Outline

- Basic concepts
- Hedgehog
- Experimentations

Basic Concepts

Data flow graph

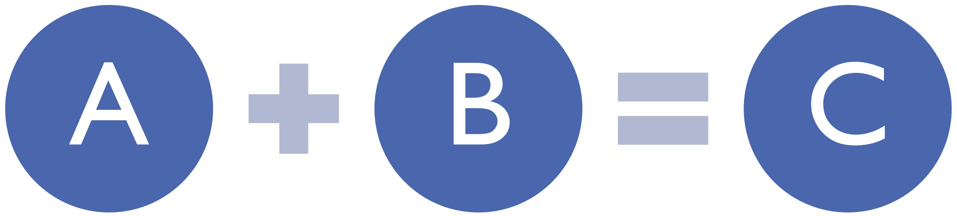
Data pipelining

HTGS & library

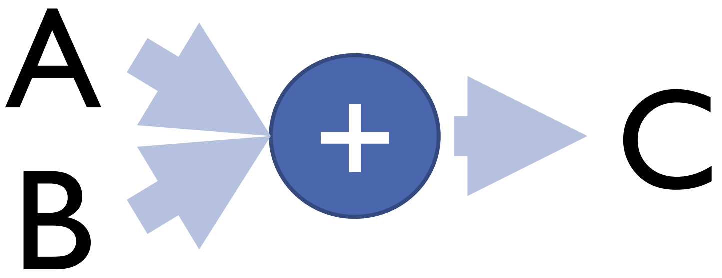
Asynchronous Data Flow Graph

- Program model
  - Directed graph representation
  - 1 entry and 1 exit point (source and sink)

- Components
  - Nodes: computations or state management
  - Edges: directed information flow



Addition algorithm

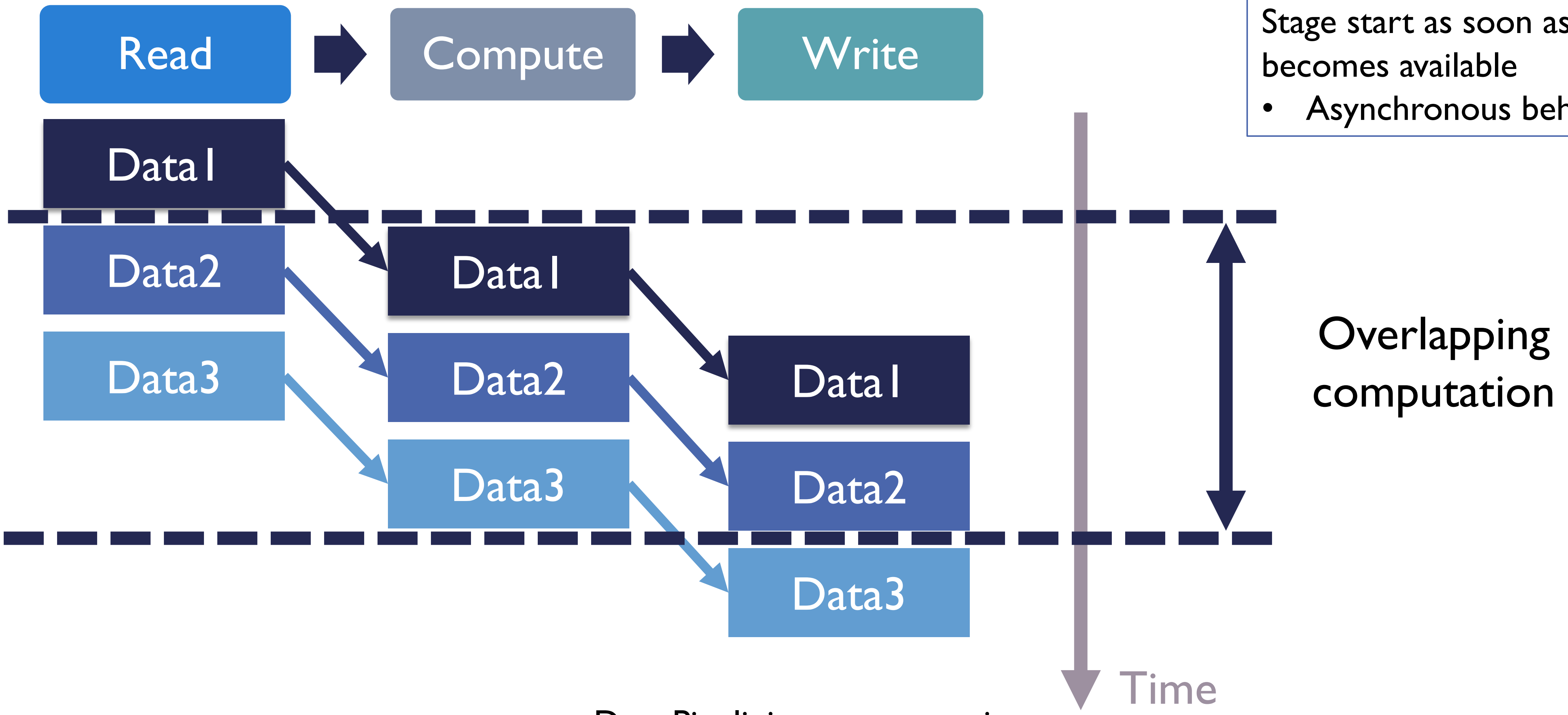


Data Flow representation

Data Pipelining

Stage start as soon as data becomes available

- Asynchronous behavior



Data Pipelining representation

Hybrid Task Graph Scheduler - HTGS

- **Coarse-Grained** Parallelism
  - **Pipelined** Multi-Threaded
  - **Multi-CPU** and **Multi-GPU**

- **C++ 11** headers-only library
  - **Visual Debugging** Feature
  - **Rich** API

Blattner T., Keyrouz W., The Hybrid Task Graph Scheduler API, (2017) GitHub repository, https:// github.com/usnistgov/HTGS

Blattner, T. et al., J Sign Process Syst (2017) 89: 457

https://doi.org/10.1007/s11265-017-1262-6

Hedgehog

Overview
API
Usage
Example

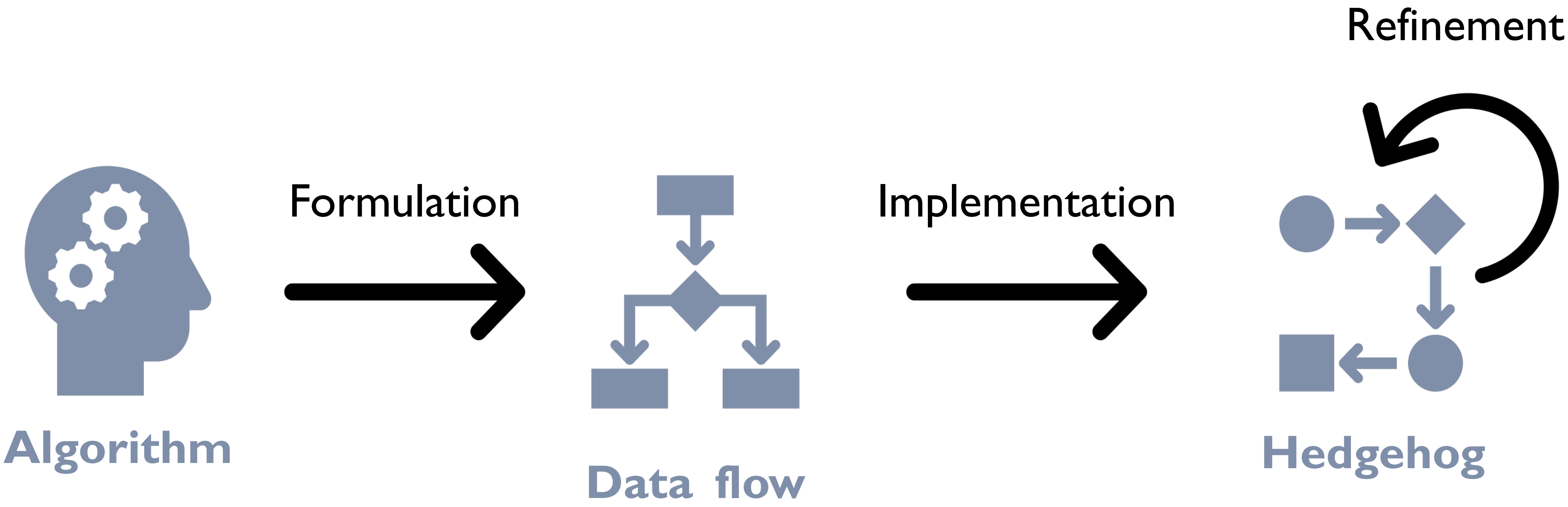
Overview

- **Coarse grain** parallelism
  - **Dataflow graph representation**
  - **Data pipelining** to obtain **performance** & keep **hardware busy**
  - **Separation of concerns:**
    - Tasks; State; Memory Management

- **C++ 17**, headers-only library
  - **General** purpose
  - **Open source and available**

- **Metaprogramming** for **type safety**

Methodology



Methodology used in Hedgehog

API - Nodes

- Multiple Inputs - Single Output
- Shutdown virtual method to break cycles

- **Tasks**
  - Step of an algorithm / **Computation kernels**
    - **Special task** for (NVIDIA) GPU computations
  - **Multithreaded**

- **State manager—single-threaded**
  - **Local** computation's **state** management
  - State shared between different managers in the graph

API - Memory Manager

- Throttles memory usage
- Links to a task or state
- Pool of available pieces of data

- Static
  - Create *n* objects calling a specific constructor
  - Ensure constructor signature by using SFINAE construct

- Dynamic
  - Create *n* objects calling default constructor

- Mechanism to recycle memory / objects

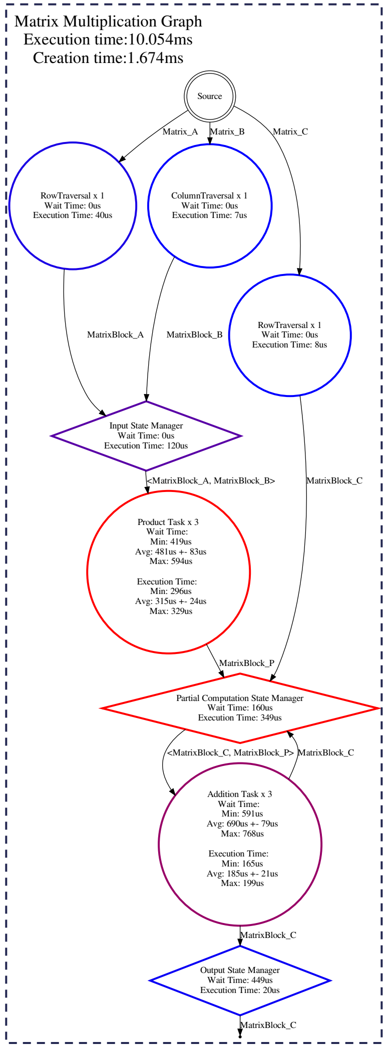
API - Graph

- Graph
  - Algorithm representation
  - Group nodes (tasks, state manager, memory manager)
  - Can be part of another graph
    - Share or compose algorithms
  - Bind a graph to a GPU
  - Only object used by an end-user

- Execution Pipeline
  - Duplicate graph
    - Data decomposition rules
    - Associate each graphs to GPUs

Explicit representation

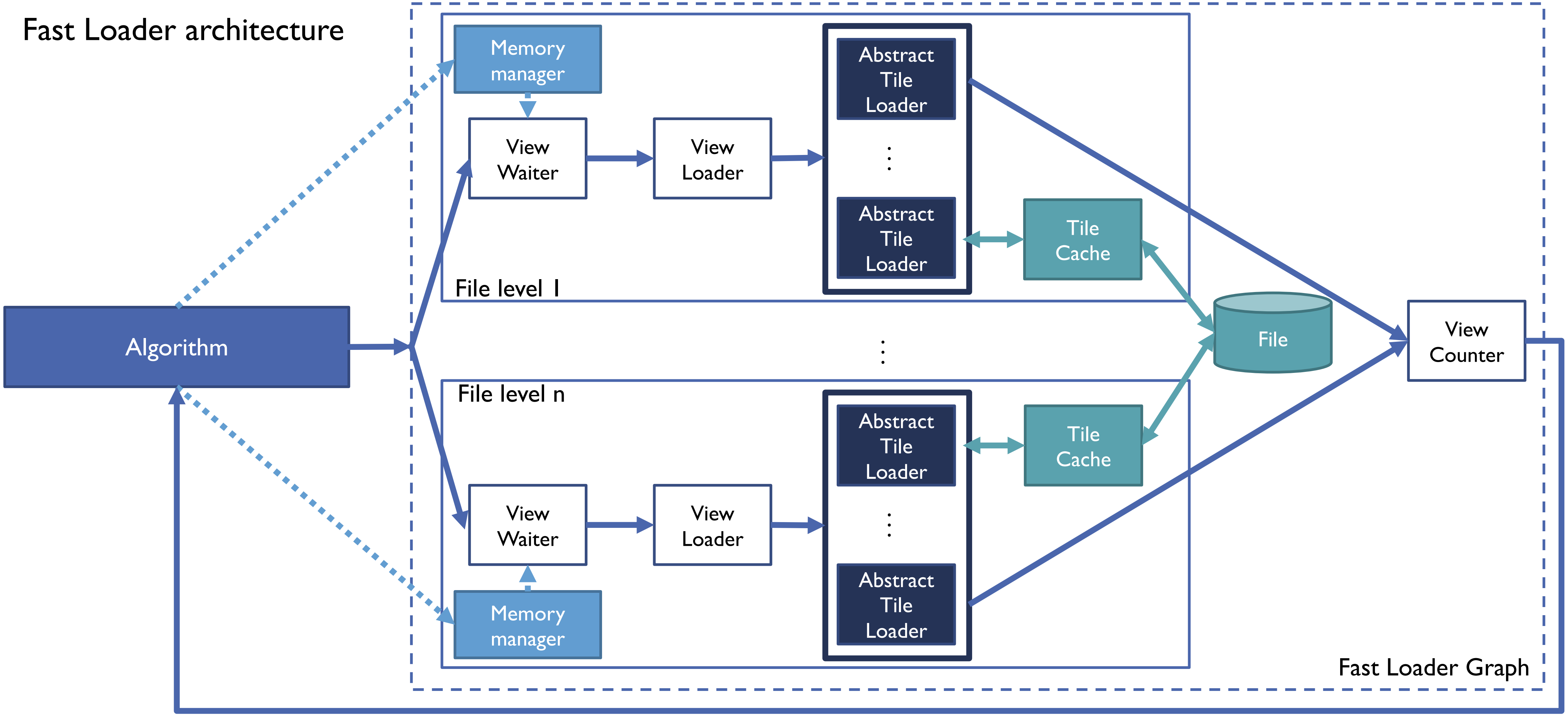
- Create a graphical representation
  - Very low overhead (task level)
- Information gathered
  - Graph: execution & creation times
  - Nodes: wait & execution times
- Node colors
  - Based on execution & wait times
- Multiple options (all threads)



Dot representation

Library Example (Fast Loader)

Fast Loader architecture



Safety @ Compile Time (Metaprogramming)

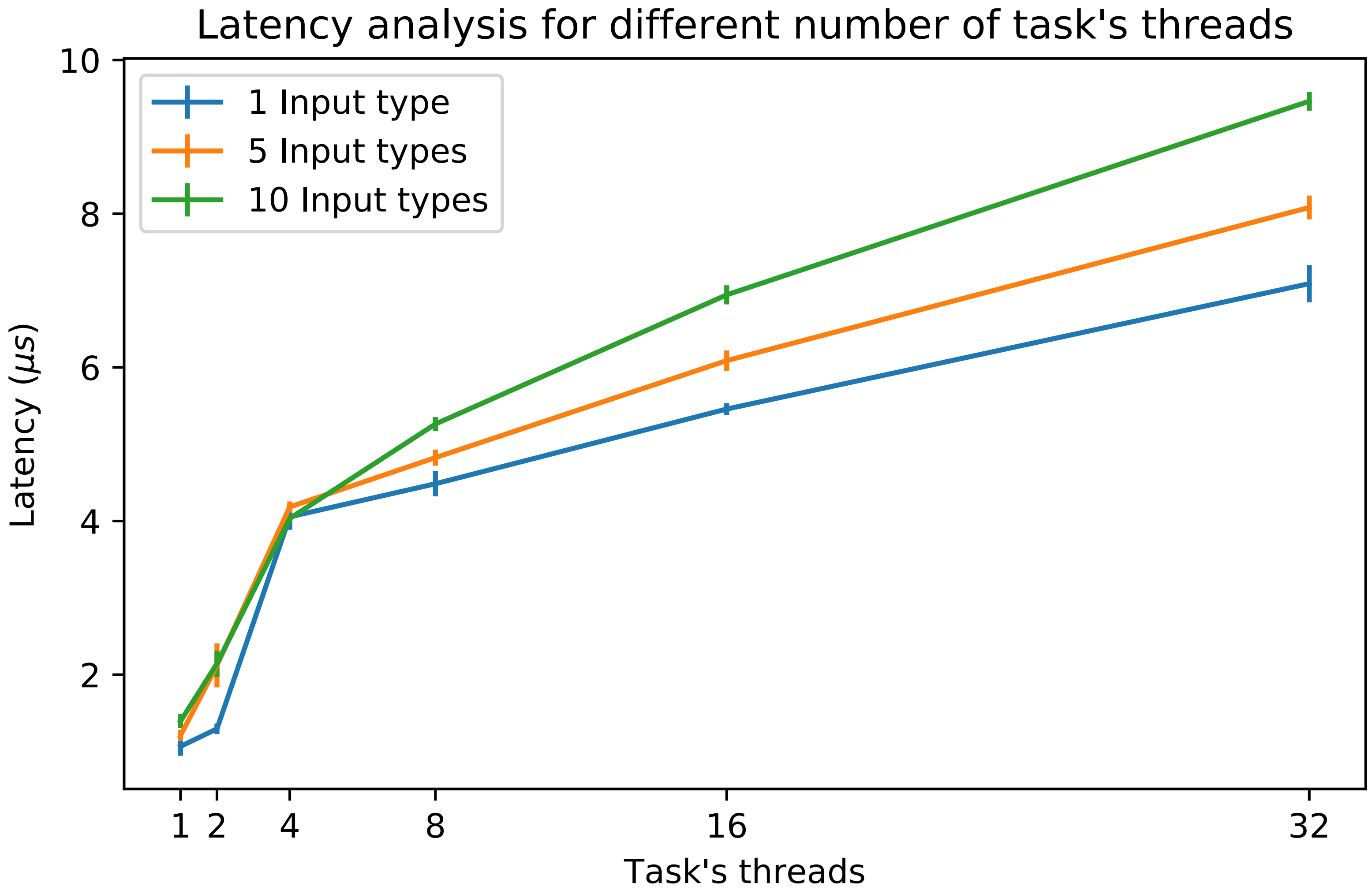
- Checks coherency rules with **traits** and **constexpr**:
  - A **graph's input task** has **at least one** of this **input type** corresponding to one of the **graph's input type**
  - **Two linked tasks** have **at least one common type**: task output's type correspond to at least to one of the other input types' task

- Checks restriction rule with **traits**:
  - To connect a memory manager to a node, **the managed type is the node's output type**

- Generates code with **SFINAE** construct:
  - Generate constructor for managed types

- Can be easily modified to take advantage of C++20

System latency



Usability (Summer 2019)

- Rising sophomore student

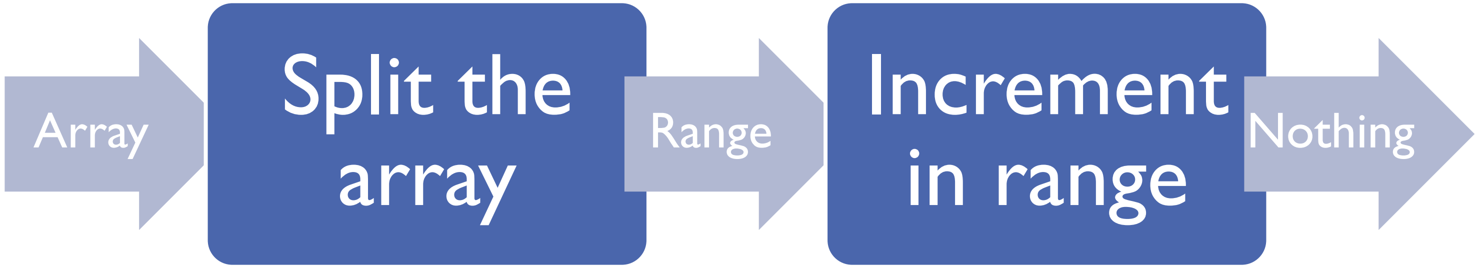
- No knowledge about
  - C++
  - Parallel programing

- In < 3 months:
  - Learned enough C++ to use the library
  - Created base graphs to represent algorithm
  - Prototyped several numerical linear algebra operations
  - Got (good) results…

Example

- Goal
  - API overview
  - Increment all elements in an array

- Algorithm
  - Split array into chunks
  - Increment chunks in parallel



Algorithm representation

Example: Some data

```
#include <hedgehog/hedgehog.h>

const size_t SIZE = 1000000000;    // 10^9 --- ginormous size

using MYARRAY = std::array<int, SIZE>;

struct ItBeginEnd {
  MYARRAY::iterator
      begin_,
      end_;

  ItBeginEnd(MYARRAY::iterator const &begin, MYARRAY::iterator const &end)
      : begin_(begin), end_(end) {}
};
```

Example: Tasks / Split vector

```
class SplitVector : public hh::AbstractTask<ItBeginEnd, MYARRAY> {
 private:
  size_t batchSize_ = 0;

 public:
  explicit SplitVector(size_t batchSize) : AbstractTask("Split Vector Task"), batchSize_ (batchSize)
{}

  void execute(std::shared_ptr<MYARRAY> v) override {
  for (size_t pos = 0; pos < SIZE; pos += batchSize_) {
      this->addResult(
        std::make_shared<ItBeginEnd>(v->begin() + pos,  v->begin() + std::min(SIZE, pos +
batchSize_)));
    }
  }
};
```

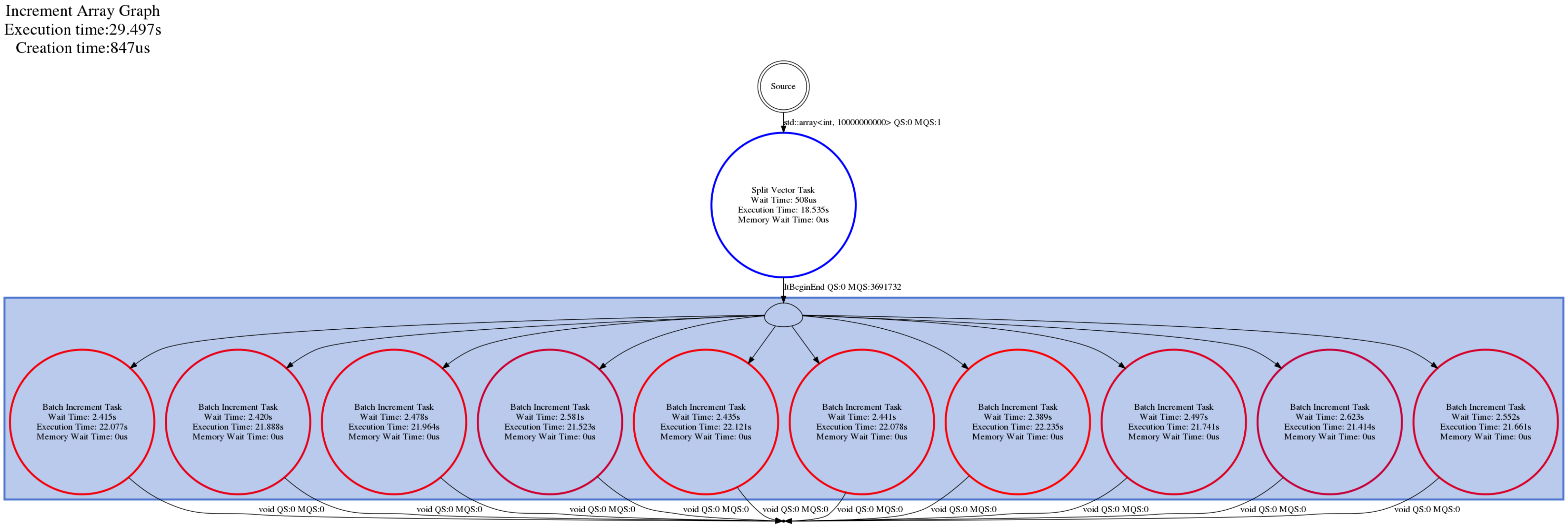
Example: Tasks / Batch Increment

```
class BatchIncrement : public hh::AbstractTask<void, ItBeginEnd> {
 private:
  size_t increment_ = 0;

 public:
  explicit BatchIncrement(int increment, size_t numberThreads)
  : AbstractTask("Batch Increment Task", numberThreads), increment_(increment) {}

  std::shared_ptr<AbstractTask < void, ItBeginEnd>> copy() override{
    return std::make_shared<BatchIncrement>(increment_, this->numberThreads());
  }

  void execute(std::shared_ptr<ItBeginEnd> ptr) override {
    std::for_each(ptr->begin_, ptr->end_, [this] (int& x) { x += increment_; });
  }
};
```

Example: main

```
int main() {
  auto  myArray = std::make_shared<MYARRAY>();
  // Instantiate graph parts
  auto graph = std::make_shared<hh::Graph<void, MYARRAY>>("Increment Array Graph");
  auto splitVectorTask = std::make_shared<SplitVector>(1000); // batchSize:1000
  auto batchIncrementTask = std::make_shared<BatchIncrement>(100, 10); // +100, 10 threads
  // Construct Graph: link tasks and set graph's input / output, and run it
  graph->input(splitVectorTask);
  graph->addEdge(splitVectorTask, batchIncrementTask);
  graph->output(batchIncrementTask);
  graph->executeGraph();
  // Send data to the graph, and wait for termination
  graph->pushData(myArray);
  graph->finishPushingData();
  graph->waitForTermination();
  // Create dot representation after computation completes
  graph->createDotFile("Test.dot", hh::ColorScheme::EXECUTION, hh::StructureOptions::ALL);
}
```

Example: Graph Representation



Algorithm dot representation

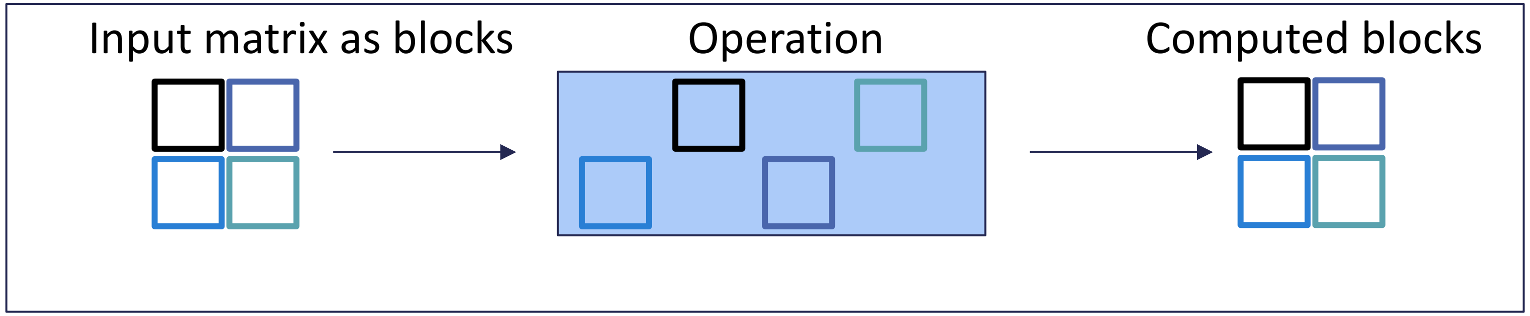
Experiments

Linear Algebra Routines

Matrix Multiplications experiments

Linear Algebra Routines - Exploiting matrix decomposition

- Matrix decomposition inside operation
  - Most linear algebra implementations take advantage of this internally
- Matrix decomposition outside operation
  - Allows for streaming mode of computation
    - Output blocks can be used immediately
    - Time for using computed data should immensely decrease
  - Not available with other numerical linear algebra libraries

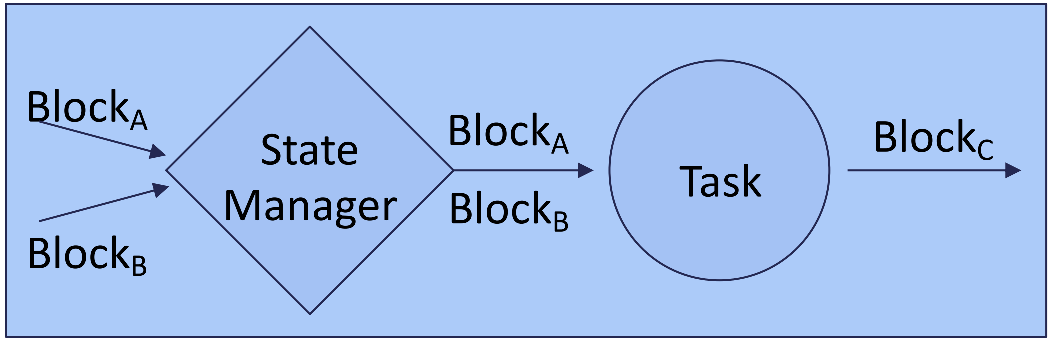


Streaming of matrix blocks in and out of an operation

Hedgehog Matrix Block Library (HMBLib)

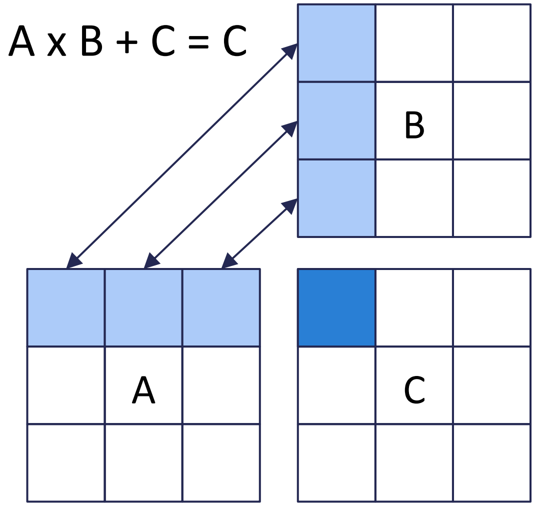
- Hedgehog - API that aids to obtain performance
  - Designed for single system with many CPU cores & multiple GPUs
- Linear algebra subroutines (graphs)
  - Tasks
  - State-Managers
    - States
- Reuse kernels from existing libraries

Example Graph (A + B = C)



Linear Algebra - General Matrix Multiplication

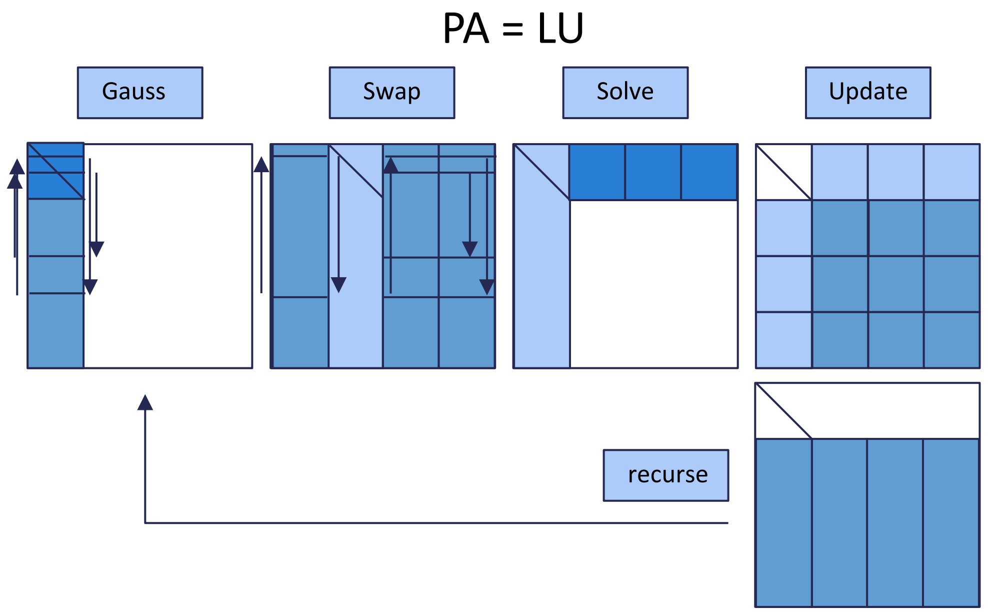
- Compatible with BLAS (gemm)
- Multiply Blocks with same inner dimension
  - Uses OpenBLAS (gemm)
- Add blocks together
  - Add sum to corresponding block of matrix C
- Output final block



Matrix Multiplication Representation

Linear Algebra - LU decomposition, partial pivoting

- Factor a matrix as the product of two triangular matrices
  - Used to solve: Ax = B
  - Compatible with LAPACK's getrf
- Recursive algorithm
- Row swapping enabled
  - Allows for more generalized matrices
  - Uses LAPACK's laswp



LU decomposition algorithm

Linear Algebra - Performance Study with HMBLib

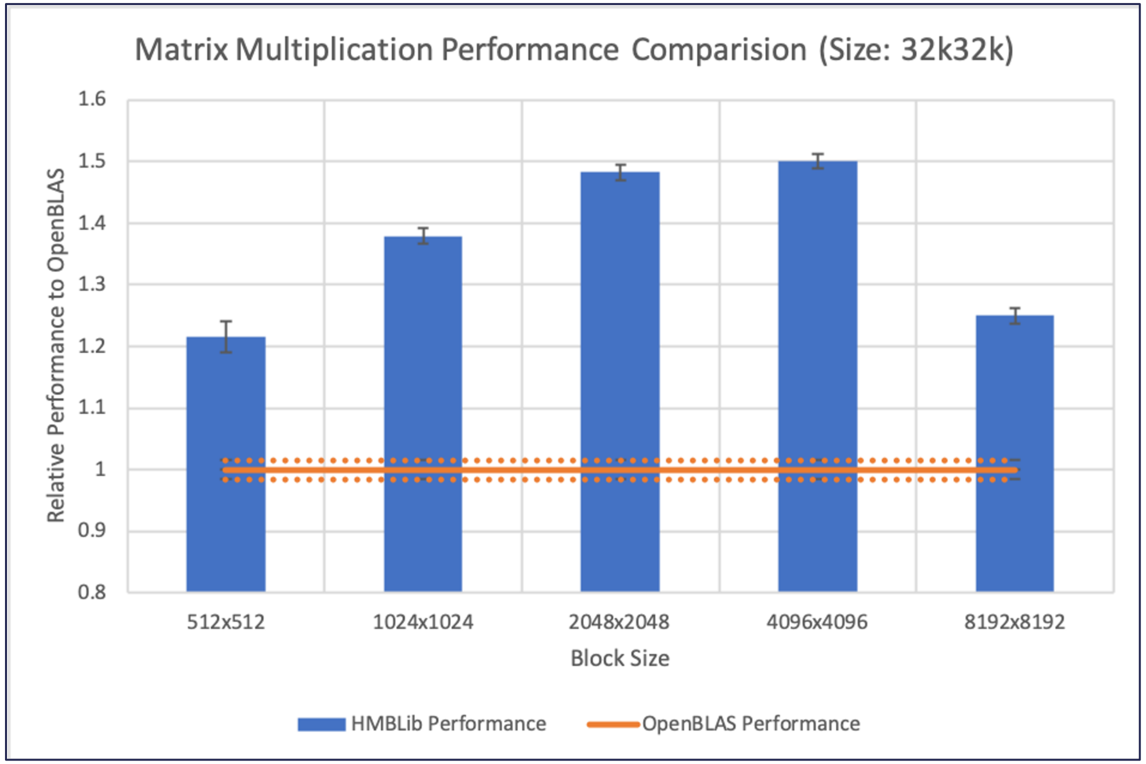
- HMBLib v. OpenBLAS (gemm) & LAPACK (getrf)
- 32,768 x 32,768 sized double precision matrices
  - Over 1 billion objects
  - ~16 GBs each

- Computer specifications for study:
  - 1 node, 2x 14 physical cores (56 logical)
    - 2 x Xeon E5-2680 @ 2.40 GHz
      - AVX2 (256-bit SIMD vector instruction)
  - 512 GB Memory

Linear Algebra - Matrix Multiplication Performance Study

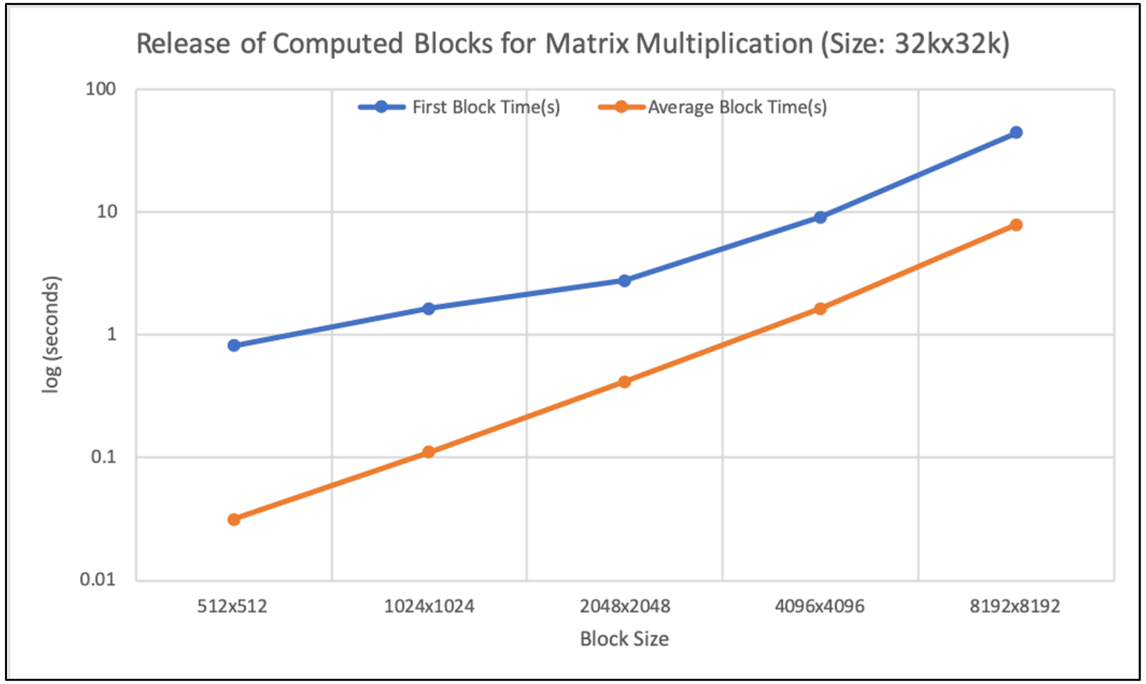
- HMBLib v OpenBLAS (gemm) overall computation comparison
  - ~660 GFlops v. ~445 GFlops
  - 1.50x performance improvement

$$\text{Performance} = \frac{\text{OpenBLAS Time(s)}}{\text{Computation Time(s)}}$$



Linear Algebra - Releasing Final Blocks (GEMM)

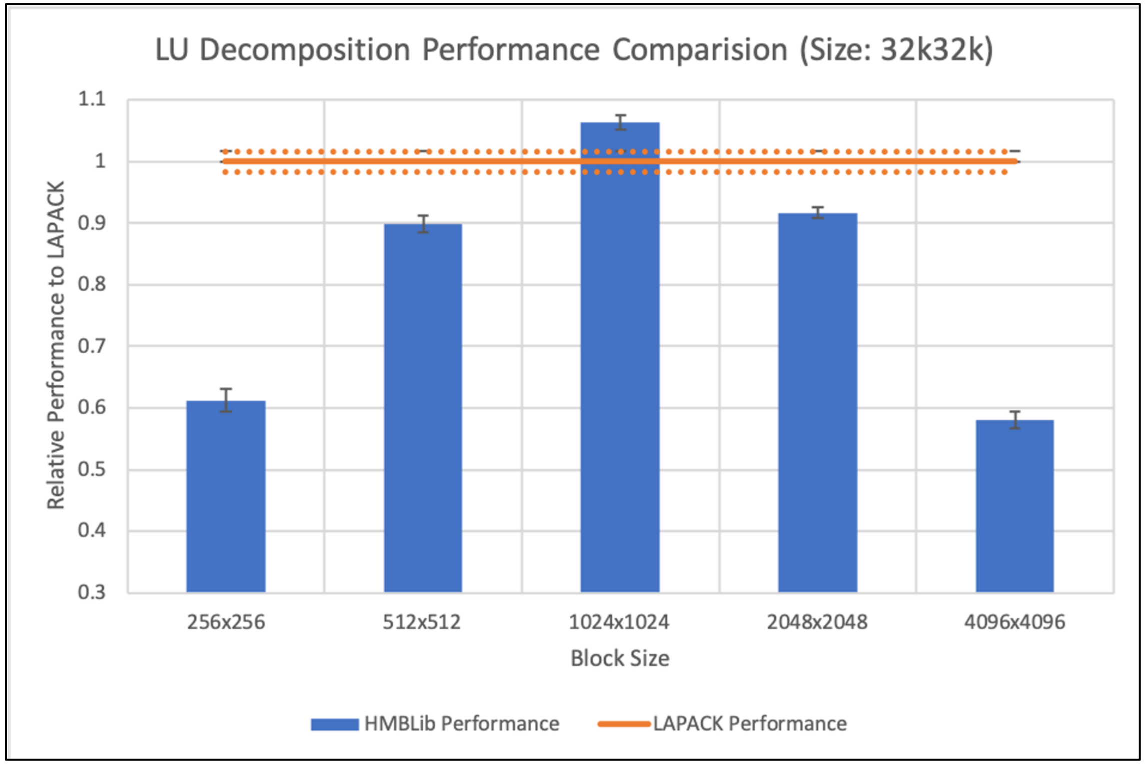
- First block time - time to release first block data
- Average block time - time to release average block data
- HMBLib vs OpenBLAS (gemm) time for first output comparison
  - 57x less for first output of computed data



Linear Algebra - LU w/PP Performance Study

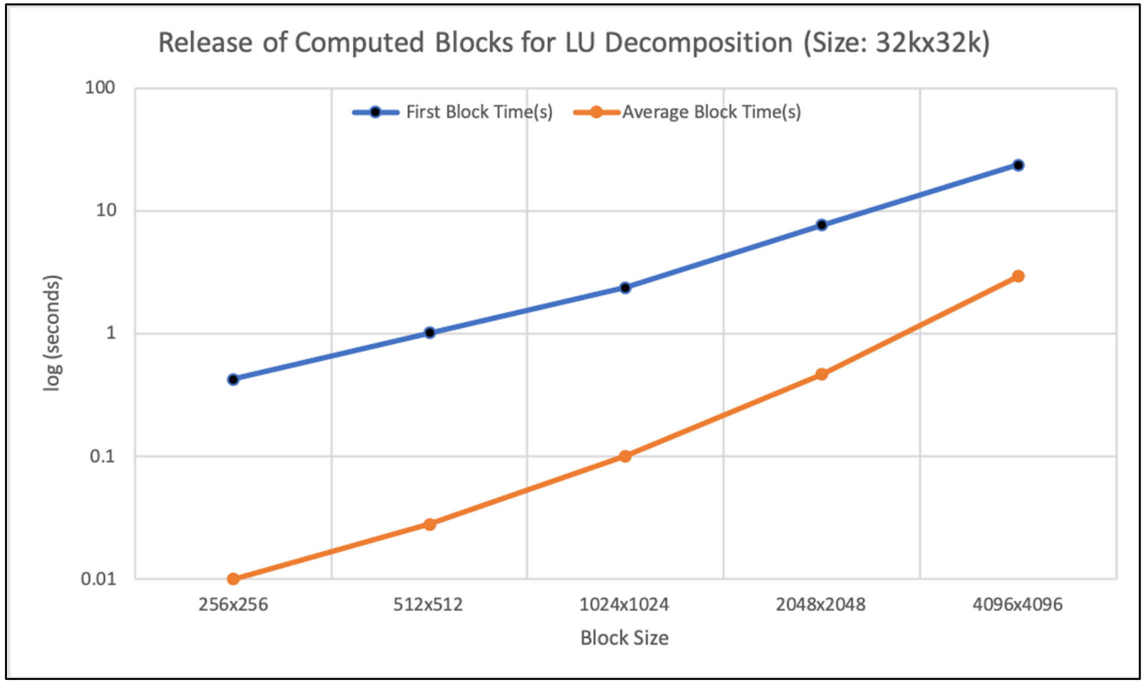
- HMBLib v LAPACK (getrf) overall computation comparison
  - ~238 v. ~224 GFlops
  - 1.06x performance improvement

$$\text{Performance} = \frac{\text{LAPACK Time(s)}}{\text{Computation Time(s)}}$$



Linear Algebra - LU w/PP Performance Study

- HMBLib v LAPACK (getrf) time for first output comparison
  - 42x less time for first computed data



Hedgehog CUDA Acceleration Experiment

- Objective:
  - Adapt Hedgehog OpenBLAS GEMM to use cuBLAS

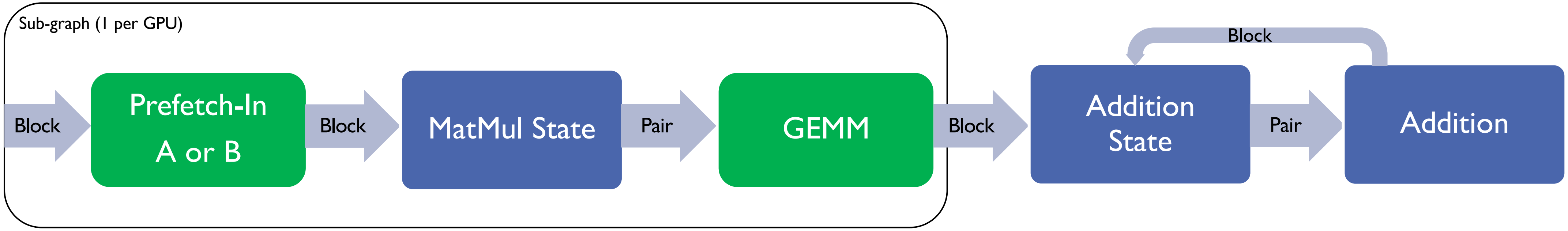
- Goals:
  - Analyze performance to observe overhead related to Hedgehog
    - Compared with cublasXT and cublasMG as baselines
  - Use CUDA optimization techniques to keep the GPU(s) busy

- Hardware:
  - SuperMicro SYS-2029GP-TR Server
    - 2x 16 core Intel Xeon Silver 4216 CPUs @ 2.1 GHz
    - 792 GB DDR4
    - 4x Tesla V100-PCIe w/ 32 GB HBM2

Hedgehog(HH)-GEMM CUDA Optimizations

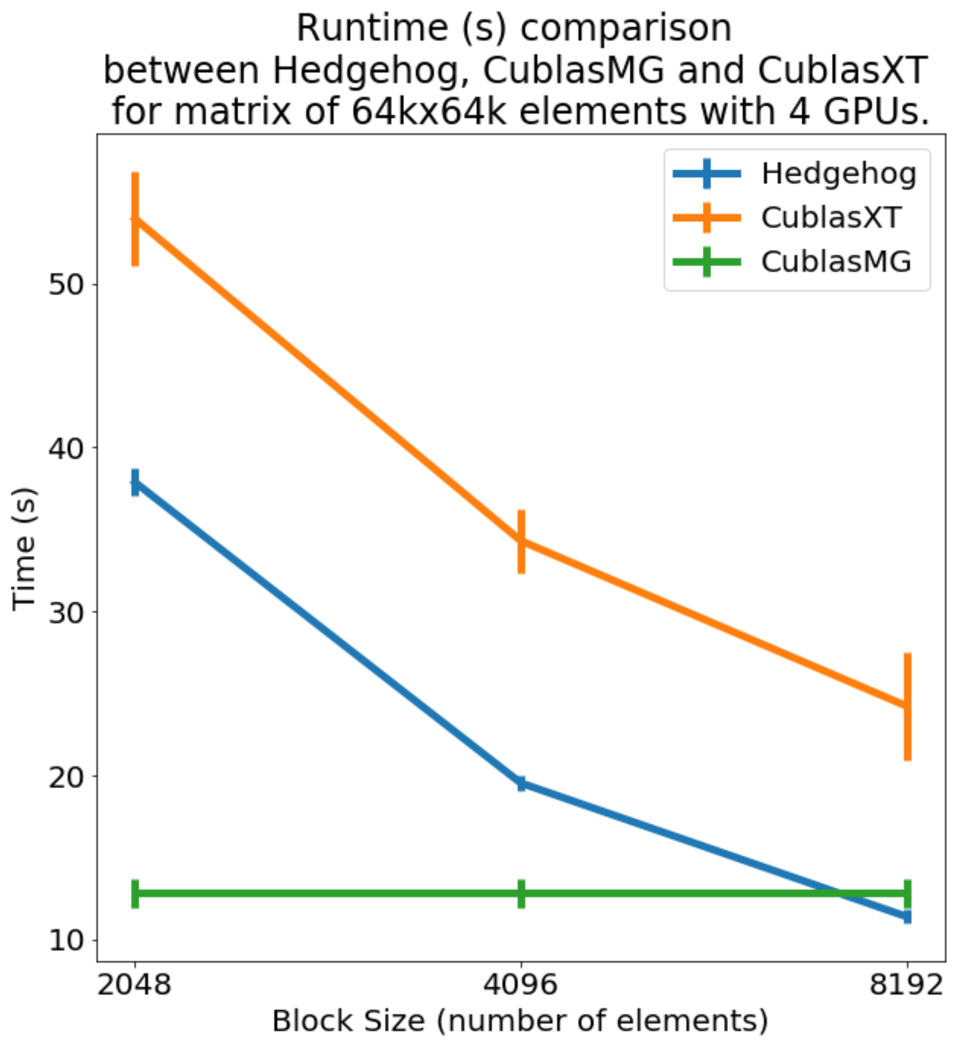
- CUDA technologies used
  - Unified memory
  - Asynchronous pre-fetch
  - Concurrent kernel execution
  - Synchronization through events
- HH-GEMM CUDA
  - Operates with user-specified block-size
  - Each block is contiguous and allocated outside of graph
    - No support for 2D cudaMemPrefetchAsync

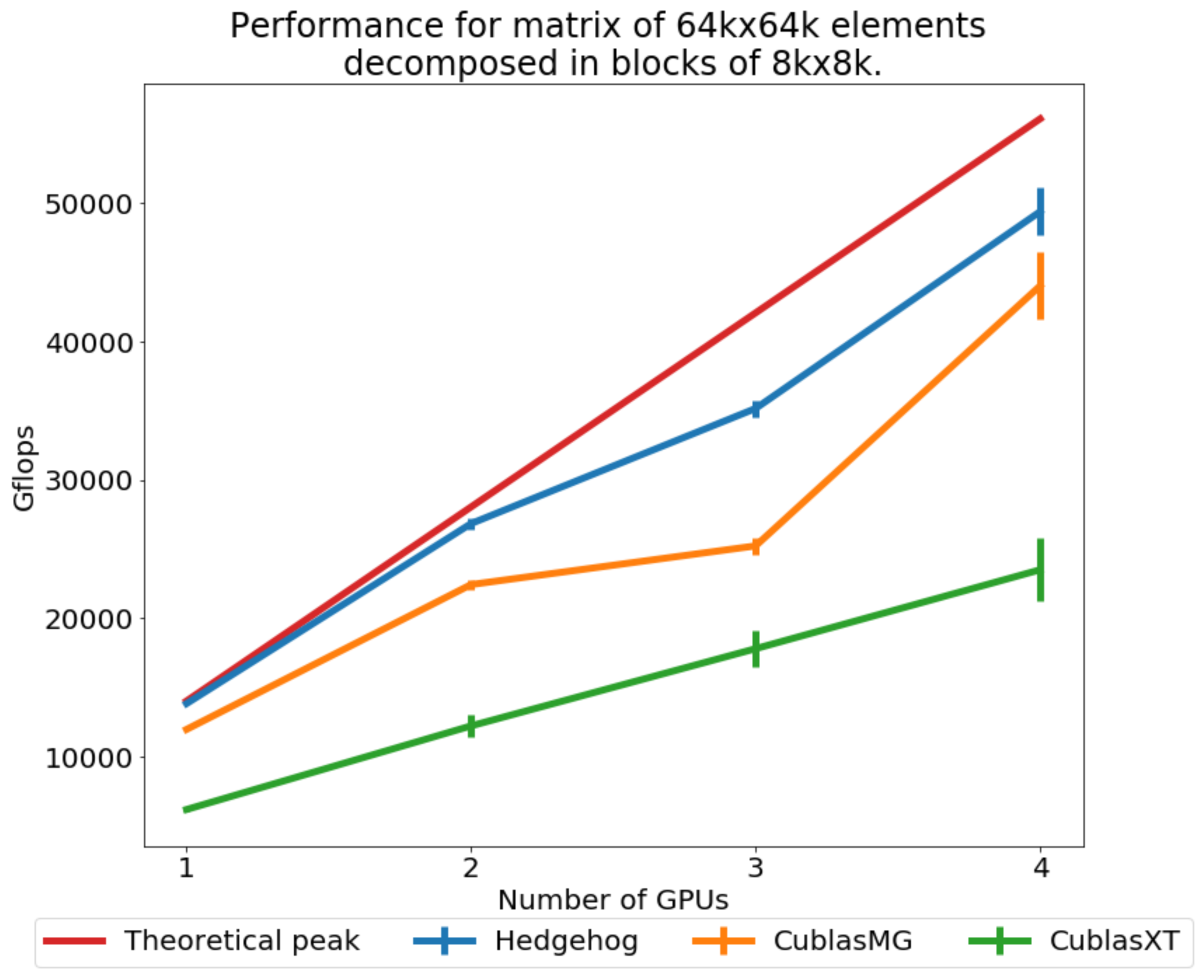
HH-GEMM CUDA Graph



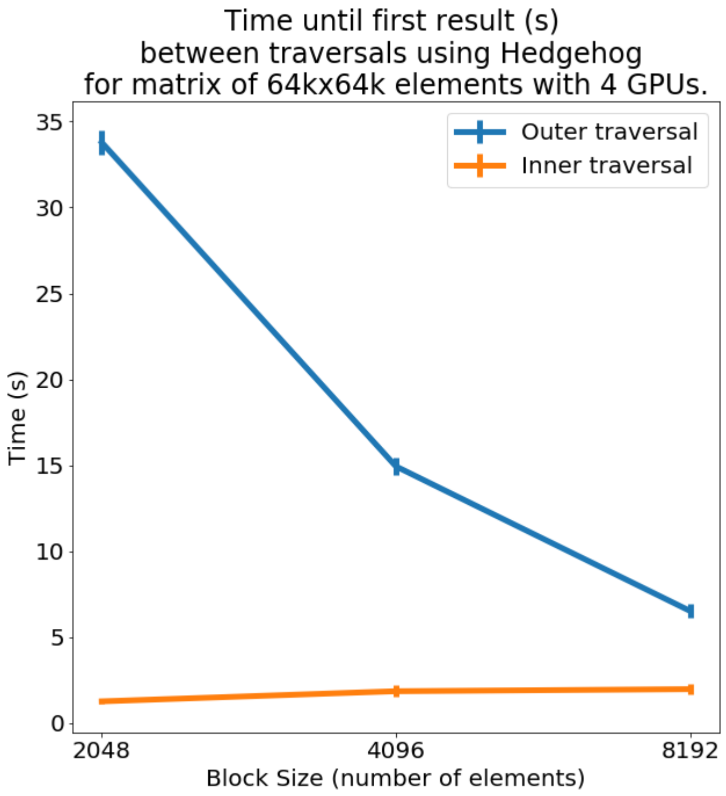
	Prefetch-In A or B	MatMul State	GEMM	Addition State	Addition
Threads	2 (1 stream per thread)	1	5 (1 stream per thread)	1	N
Functionality	HH Get $\text{Mem}_{A \mid B}$ Prefetch $\text{Mem}_{A \mid B}$ CPU→GPU Create Event_1	Pair Mem_A and Mem_B (based on MatMul)	HH Get $\text{Mem}_{\text{Partial(P)}}$ Prefetch Mem_P CPU→GPU Synchronize Event_1 cublasSgemm(Mem_P ,Mem_A ,Mem_B) Synchronize Stream Recycle $\text{Mem}_{A \,\&\, B}$ Prefetch Mem_P GPU→CPU Create Event_2	Pair Mem_P with C	Synchronize Event_2 C = Mem_P + C Recycle Mem_P

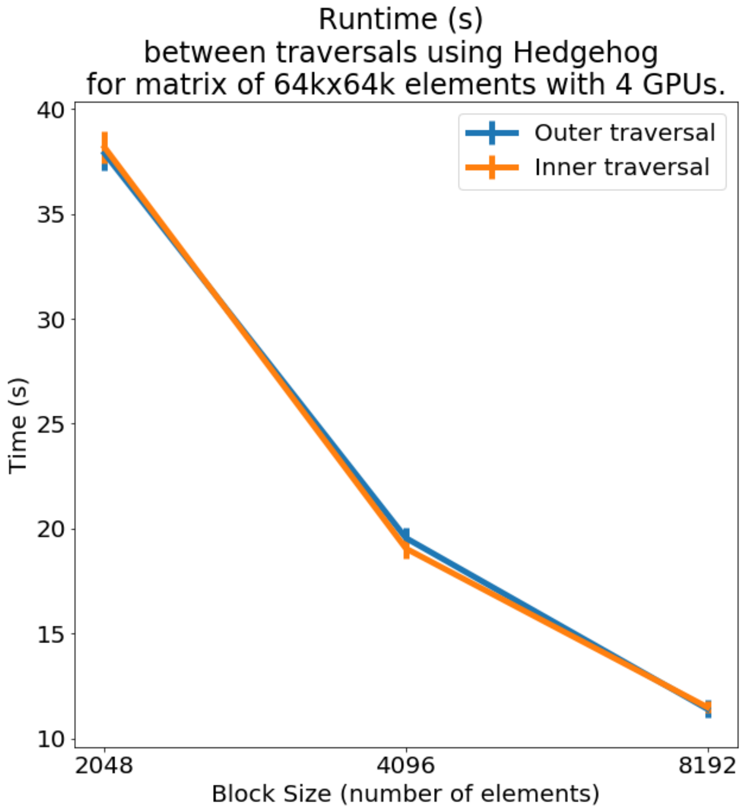
HH-GEMM CUDA Results 16 GB Size Matrices





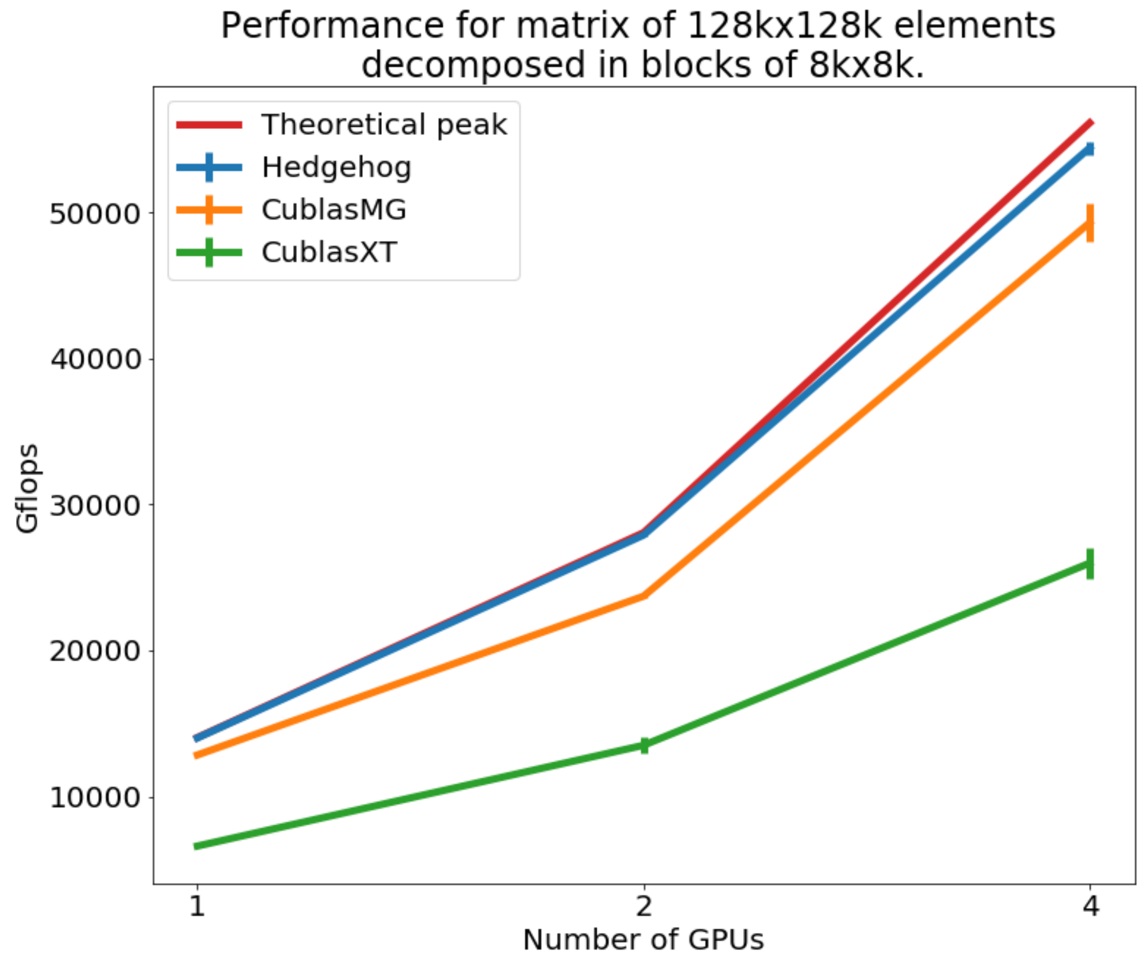
Streaming Linear Algebra with HH-GEMM CUDA

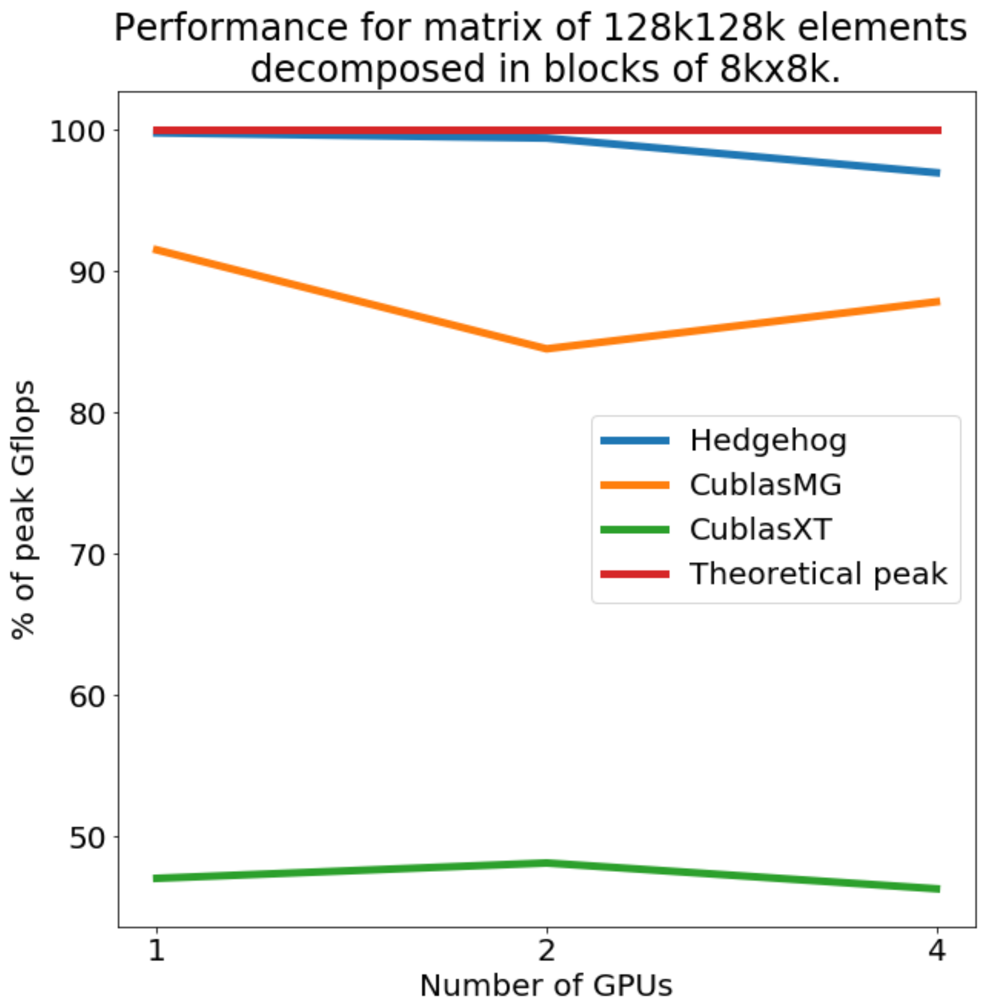




- Streaming linear algebra
  - Required minor modifications to code to switch between inner/outer traversals
    - Change loop order for pushing block data into graph
    - Alter memory pool size to have sufficient memory for both A and B
- Performance can be detrimental if there is insufficient GPU memory (unified memory paging)

H-GEMM Results 64 GB Size Matrices





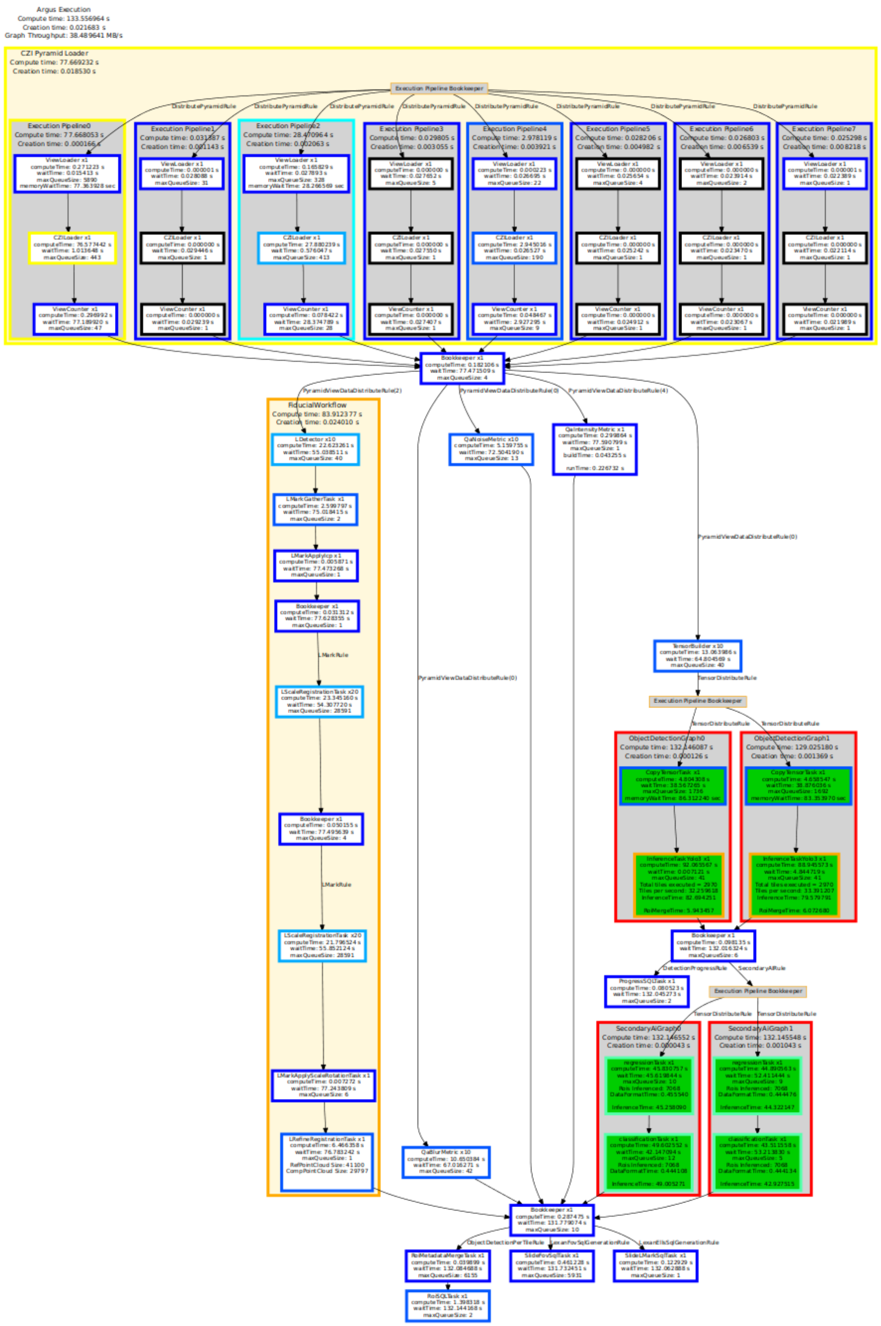
Users

NIST
Full Slide Microscopy Analysis

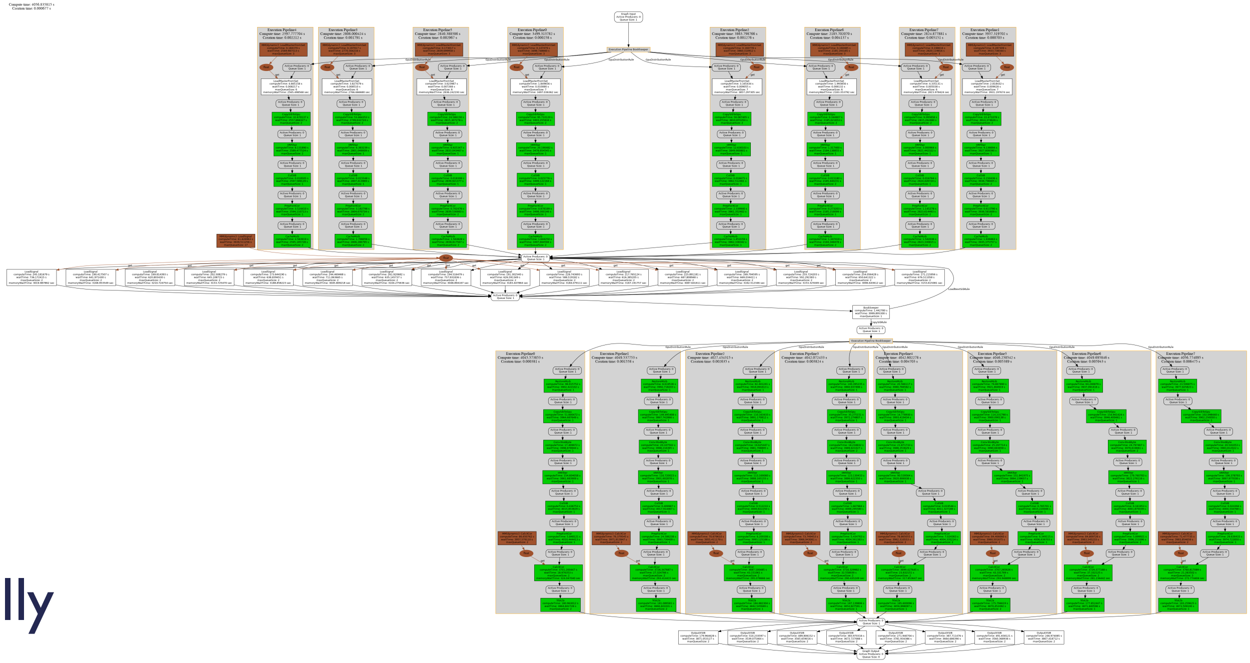
- Processing Hardware:
  - 2x - Xeon Gold 5120 “Skylake” 14-core CPUs
  - 2x - NVIDIA GTX Titan V graphics cards

- 100,000 x 50,000 pixel images
  - Traditional computer vision
  - Inference using TensorRT
    - Object Detection (Yolo V3)
    - Classification (Resnet50)

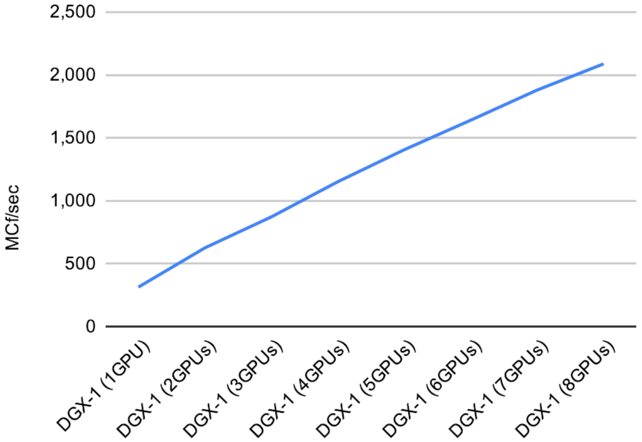
- End-to-end 60-90 seconds
  - Scales to number of GPUs



Comprehensive Nuclear-Test-Ban Treaty Preparatory Commission



- Processing Hardware:
  - DGX-1 server (8xV100s)
- Monitors the nuclear test ban treaty
  - 300+ stations with 1000+ sensors globally
- 2.268 billion cross correlations per second
  - 8 GPUs
  - Scales with number of GPUs



Conclusion

Which library allows us to manage a node with a lot of threads and one or multiple GPU, with an explicit representation of an algorithm (that exists during execution), and a high-level abstractions (without loss of potential performance) ?

- **Hedgehog**
  - Based on an explicit **Data Flow Graph** using **Data Pipelining**
  - With a costless feedback that allows refinement
- **HMBLib**
  - **Concept of streaming data shows promise**
    - Relevant for GPU and CPU computation
  - Potential Applications: Large image processing, Galaxy and space mapping
- **Available**
  - **Hedgehog:** https://github.com/usnistgov/hedgehog
  - **Tutorials:** https://pages.nist.gov/hedgehog-Tutorials, https://github.com/usnistgov/hedgehog-Tutorials

Future

- Experiments with extending Hedgehog to operate beyond a single node
- General purpose libraries based around Hedgehog
  - Streaming full-slide microscopy analysis
- Compile-time static graph analyses
  - Check race conditions
  - Deadlock
- Principled dataflow-based "code generation"
  - Automated rule generation

Thank you

Any questions ?